Loop Parallelisation for the Jikes RVM

Jisheng Zhao, Dr. Ian Rogers, Dr. Chris Kirkham, Prof. Ian Watson
The University of Manchester
{jisheng.zhao, ian.rogers, christopher.kirkham, ian.watson}@manchester.ac.uk

Abstract

Increasing the number of instructions executing in parallel has helped improve processor performance, but the technique is limited. Executing code on parallel threads and processors has fewer limitations, but most computer programs tend to be serial in nature. This paper presents a compiler optimisation that at run-time parallelises code inside a JVM and thereby increases the number of threads. We show Spec JVM benchmark results for this optimisation. The performance on a current desktop processor is slower than without parallel threads, caused by thread creation costs, but with these costs removed the performance is better than the serial code. We measure the threading costs and discuss how a future computer architecture will enable this optimisation to be feasible in exploiting thread instead of instruction and/or vector parallelism.

1. Introduction

Parallelising compilers, sometimes known as supercompilers, will soon be common place with GCC 4, where support comes in the form of vectorisation [12]. Vectorisation is used to produce code for the single-instruction multiple-data (SIMD) instruction sets of modern multimedia architectures. An alternative form of parallelisation to vectorisation is thread based. Threads can execute on separate computing resources and hence in parallel with each other.

Vectorisation requires analysis of inner loops of code and to determine if they can be made to work on several data operands at the same time. Parallelisation can work on any loop of code, but the cost of thread creation and completion detection mean it is best performed on outer loops [9, 16]. Outer loops invariably have more complex data dependencies within them, and thus detecting loops to parallelise often isn't possible. Consequently current microprocessors offer good vector support and limited support for the threading model.

In this work we take the view that vectorisation, as with ILP, will become limited and that a better compromise in the design balance may be for more parallel execution units (i.e. more cores in a chip multi-processor) rather than extending hardware support for vectors and ILP. We present a new implementation of the well-known and simplistic DOALL parallelisation compiler optimisation [9, 16]. Uniquely for our system we implement this in the environment of a JVM, to allow for parallelisation to occur at run-time and, compared to prior JVM oriented systems, without the involvement of the programmer.

Our work assumes that all loops are best parallelised into threads, a technique which, as described above, is overly eager for current microprocessors. However, if the cost of maintaining a thread were brought close to the cost of farming work to vector processing units then we believe threads are to be preferred to vectors as the parallel execution units are general purpose. This is the model currently proposed by our Jamaica architecture [1]. However, it may ultimately be likely that the best situation is a compromise of vector and parallel resources in a heterogeneous chip multiprocessor environment.

This paper is split into 5 further sections. Section 2 describes an initial set of loop optimisations used to make Java bytecode amenable to parallelisation. Section 3 describes the loop parallelisation optimisation itself. We perform a performance analysis of thread creation and completion costs, and then measure our performance, the current performance and the performance without thread costs, in Section 4. Section 5 discusses runtime parallelisation, as has been demonstrated in this paper, motivating future computer architecture and compiler research. Section 6 concludes the paper.

2. Making Java Loops Parallelisable

The existing work such as javar, High Performance Fortran and OpenMP have allowed programmers to express which loops were amenable to thread parallelisation and the compiler could forget dependence analysis [3, 7, 4]. Substantial research work has shown how programs can be transformed by the compiler and better exploit available parallelism. This work has included automatic parallelisation, that has no need for programmer intervention or compiler constructs that imply dependencies can be ignored. GCC 4 fits this criterion, as do Intel's Fortran/C compiler and Matlab*P [12, 2, 6].

These previous solutions have looked at statically determining parallelism and then creating binaries to utilise it. We propose using similar techniques in the dynamic compilation environment of Java - specifically the Jikes Research Virtual Machine (RVM) [8]. Dynamic compilation enables run-time feedback to guide where optimisation could be performed. The loops we are looking for are the simplest form to parallelise, DOALL amenable loops. These loops have no loop carried dependencies. So we could consider parallelising an array *fill* routine such as this from the GNU Classpath library's *java.util.Arrays* implementation:

```
for (int i = fromIndex; i < toIndex; i++)
    a[i] = val;
```

Figure 1. DOALL **amenable memory assignment loop**

Java adds a complication to dependency analysis - exceptions. The loop above when in the internal form of the Jikes RVM has guards added that capture these dependencies and implicitly adds exception edges to the control-flow graph (CFG).

IEEE COMPUTER SOCIETY

The resulting loop with the exception dependencies captured looks like the one below:

```
for (int i = fromIndex; i < toIndex; i++) {
   g1 = null_check a;
   g2 = bounds_check a, g1;
   g3 = guard_combine (g1, g2);
   a[i] = val, g3;
}
```

Figure 2. Previously DOALL amenable memory assignment loop after the introduction of exceptions

The compiler can use code motion to move the null check out of the loop, but the array bounds check must be performed on each iteration. We have solved this problem for the loops we are interested in parallelising using explicit guard tests and loop duplication. This technique is presented in Section 2.2, but first we present the auxiliary intermediate form used to capture loop dependencies for the loop duplication and the parallelisation optimisations.

2.1. Annotated Loop Structure Trees

Loops are detected by a back-edge in the CFG [11]. Our auxiliary form defines the header to be the basic block, reachable from outside of the loop, that has a back-edge to it. We similarly define the exit basic block to be the block that has the back-edge to the loop header or that terminates the loop. The loop header and exit basic blocks may be the same basic block.

After determining the loop header and exit basic blocks, we inspect the exit block's branch instruction. The branch instruction has two parameters and a condition. The parameters are inspected to see which vary within the loop body, as shown in Table 1.

	Parameter 1 invariant	Parameter 1 variant
Parameter 2 invariant	Single or infinite loop, not considered for optimisation.	1st parameter is the *loop iterator*, 2nd parameter is the *terminating value* for the loop.
Parameter 2 variant	2nd parameter is the *loop iterator*, 1st parameter is the *terminating value* for the loop.	Loop too complex to optimise.

Table 1. Loop parameters and the implication for the loop bounds.

We then inspect the value of the loop iterator coming into the loop to determine its *initial value*, and inspect the assignment to the iterator within the loop to determine a loop stride. When a loop doesn't match this format then we don't optimise it and leave the intermediate representation unchanged. We see in the future extensions to this form, for example, to capture break-out paths.

The nested nature of loops allows for their representation as trees. *Loop structure trees* (LSTs) are already present within the Jikes RVM; to differentiate our form we call them *annotated LSTs*. Their form proves convenient for recursively applying our transformations.

Our optimisations are performed on the Extended Array SSA form [5] that has the following key features:

1. static single assignment (SSA) form means that all variables are assigned once, guaranteeing no output or anti dependencies. Dataflow confluences are captured in the intermediate representation as special instructions known as *phi* instuctions.
2. array form extends SSA form to consider memory accesses to arrays. *phi* instructions capture the *non-killing* nature of array stores (ie. two array stores may or may not be to the same location), arrays of different types are considered to be of different *heap types* meaning they are always different locations in memory.
3. the extended nature of the form comes from its application to Java. Specifically it means the form can determine that fields within different classes are definitely the same or different, and it captures exception dependencies so that code can't be re-ordered around them.

The structure of a recognised loop is shown in Figure 3.

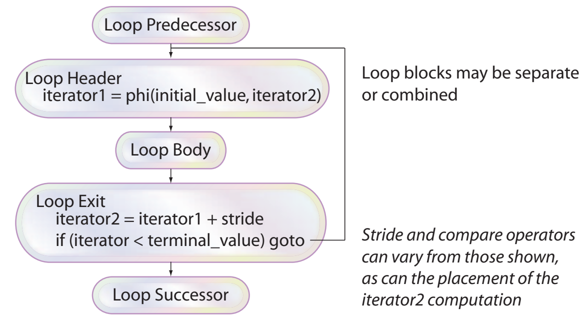

Figure 3. Loop structure for an annotated LST node

Using Extended Array SSA form means our loop analysis is simplified from considering many different kinds of dependence structures within the loop. The drawback is that the form requires construction (wasteful of time) and careful maintenance. As such our optimisations are only enabled during the compilation of hot methods. For off-line compilers such as those described in 9, the cost of generating the SSA form isn't important.

2.2. Loop Duplication

Having a fuller picture of a loop enables us to perform loop duplication, to remove null and bound checks, as well as different forms of loop unrolling. We concentrate here on loop duplication and disable loop unrolling optimisations before parallelisation. Results of using the annotated LSTs for loop unrolling have been presented in 17.

The loop duplication transformation works up from the leaves of the annotated LST. For each loop, that is in the form presented in Section 2.1, it firstly collects all array accesses and their associated guards. If no guards are present then the optimization does nothing, otherwise the following stages are performed:

1. **creation of explicit tests** - we create explicit tests for the null and bound check instructions we hope to eliminate. Bound check instructions can be eliminated from the loop if they are constant or related to the loop iterator (ie the loop iterator itself or a fixed offset from it). The guard values defined by these branch tests are recorded in a *map* against the original null and bound check guards. These are used to capture the dependency between the eliminated checks and these explicit tests, and avoid illegal code restructuring.
2. **creation of phi instructions** - two loops will be created from the original. These two loops will both define variables. To keep the SSA form we create new variables to be

IEEE COMPUTER SOCIETY

defined in the loops. The original variable's assignment is altered so that it is a phi instruction merging the definitions that will come from the two loops.

3. **creation of two duplicate loops** - we duplicate the original loop twice to avoid altering dependencies on the basic blocks for loop entry and exit. One of the duplicated loops is generated without the guards that can be explicitly tested, its guards are looked up from the *map* recorded earlier.
4. **linking of basic blocks and eliminating the original loop** - the original loop has its instructions removed, the loop header goes to the explicit null and bound checks which if they all pass goes to the loop without guards, otherwise the tests branch to the loop that will check for exceptions. The exit from the duplicated loops is to the block containing the phi instructions which in turn goes to the loop exit block.

The loop presented in Figure 1 is shown in the Jikes RVM's internal representation in Figure 4; after loop duplication the internal representation is shown in Figure 5.

```
[...]
19              LABEL4 Frequency: 8.999998
-9              phi               t30pi(I) = 0, BB3, t31pi(I), BB4
19      G       yieldpoint_backedge
23      EG      bounds_check      t26v(GUARD) = l0pa([Z,d), t30pi(I), t2pv(GUARD)
23              guard_combine     t27v(GUARD) = t2pv(GUARD), t26v(GUARD)
23              byte_astore       l1pi(Z,d), l0pa([Z,d), t30pi(I), <mem loc: array
                                  <BootstrapCL, Z >[]>, t27v(GUARD)
24              int_add           t31pi(I) = t30pi(I), 1
30              int_ifcmp         t32v(GUARD) = t31pi(I), t3pi(I), <, LABEL4,
                                  Probability: 0.9
-1              goto              LABEL6
-1              bbend             BB4
[...]
```

Figure 4. Potential DOALL amenable loop prior to duplication

The original loop, shown in Figure 4, is a single basic block in the Jikes RVM *High-level Intermediate Representation* (HIR) this is basic block 4 and is consequently the header and exit block of the loop. The loop iterator is t30/t31, which has an initial value of 0 and a terminal value constrained by a less-than on t3 (which is loop invariant). In Figure 5, the duplicated loops are in basic blocks 9 and 11, the phi instructions are in the original exit basic block 4. The explicit test is performed in basic block 12, it checks that t3 is greater than array length of the array that is part of the bounds check. The explicit test is the guard controlling the array assignment in the DOALL amenable loop in basic block 11.

Eliminating null and bound checks in this way allows for a performance improvement of up to 3% in the Spec JVM benchmarks [17, 14]. However, for one benchmark the extra analysis results in a 1.5% slow down. The overall performance improvement of annotated LST node based loop optimisations, without loop parallelisation, is 0.25%. We believe work on the adaptive compilation system will further improve the optimisation.

3. Loop Parallelisation

Figure 5 shows a loop with no dependencies that would inhibit loop DOALL parallelisation. To recognise this loop the annotated LST is recreated as described in Section 2.1. The tree is traversed from the root down in order to generate parallel outer loops. The intermediate form is first checked that no exception throwing or method calls are present within the loop. We also don't parallelise certain methods in the Jikes RVM garbage collection and threading mechanisms. Next the dependencies

```
[...]
-16             LABEL12 Frequency: 8.999998
-1              arraylength       t43i(I) = l0pa([Z,d), t2pv(GUARD)
-1              int_ifcmp         t36v(GUARD) = t3pi(I), t43i(I), >U, LABEL8,
                                  Probability: 0.00999999
-1              bbend             BB12

-16             LABEL13 Frequency: 8.999998
-1              goto              LABEL10
-1              bbend             BB13

19              LABEL4 Frequency: 8.999998
-1              phi               t30pi(I) = t39i(I), BB9, t40i(I), BB11
-1              phi               t26v(GUARD) = t35v(GUARD), BB9, t36v(GUARD), BB11
-1              phi               t27v(GUARD) = t37v(GUARD), BB9, t38v(GUARD), BB11
-1              phi               t31pi(I) = t39i(I), BB9, t40i(I), BB11
-1              phi               t32v(GUARD) = t41v(GUARD), BB9, t42v(GUARD), BB11
-1              goto              LABEL6
-1              bbend             BB4
[...]

-16             LABEL8 Frequency: 8.999998
-1              bbend             BB8

19              LABEL9 Frequency: 8.999998
-9              phi               t33i(I) = 0, BB8, t39i(I), BB9
19      G       yieldpoint_backedge
23      EG      bounds_check      t35v(GUARD) = l0pa([Z,d), t33i(I), t2pv(GUARD)
23              guard_combine     t37v(GUARD) = t2pv(GUARD), t35v(GUARD)
23              byte_astore       l1pi(Z,d), l0pa([Z,d), t33i(I), <mem loc: array
                                  <BootstrapCL, Z >[]>, t37v(GUARD)
24              int_add           t39i(I) = t33i(I), 1
30              int_ifcmp         t41v(GUARD) = t39i(I), t3pi(I), <, LABEL9,
                                  Probability: 0.9
-1              goto              LABEL4
-1              bbend             BB9

-16             LABEL10 Frequency: 8.999998
-1              bbend             BB10

19              LABEL11 Frequency: 8.999998
-9              phi               t34i(I) = 0, BB10, t40i(I), BB11
19      G       yieldpoint_backedge
23              guard_combine     t38v(GUARD) = t2pv(GUARD), t36v(GUARD)
23              byte_astore       l1pi(Z,d), l0pa([Z,d), t34i(I), <mem loc: array
                                  <BootstrapCL, Z >[]>, t38v(GUARD)
24              int_add           t40i(I) = t34i(I), 1
30              int_ifcmp         t42v(GUARD) = t40i(I), t3pi(I), <, LABEL11,
                                  Probability: 0.9
-1              goto              LABEL4
-1              bbend             BB11
```

Figure 5. Loop duplication resulting in a DOALL amenable loop at basic block 11

are examined to see if they are amenable to DOALL parallelisation. We look to see what phi nodes are present in the loop. Phi nodes for the loop iterator are expected. For loop-carried heap dependencies, we examine the array indexes to ensure the distances from the loop iterator are all the same. If all loop-carried heap dependences have the same dependence distance then the loop is amenable to DOALL parallelisation [15]. In Figure 5 the auxiliary extended array part of the SSA form isn't shown, however, the dependence distance of the heap phi nodes is 0 or equal to the loop iterator. Therefore the loop is amenable to DOALL parallelisation.

Once an amenable loop is found the following operations are performed to convert it into its parallel form:

1. **creation of loop worker class** - each parallel thread runs a loop worker that is created from the regular Java thread implementation. The loop worker has a number of fields created to hold onto invariant values used within the loop body.
2. **copying of loop body to loop worker method** - the loop body is copied in HIR form into the loop worker method. Accesses to loop invariants are modified to retrieve fields from the class.
3. **compilation of the loop worker** - as the loop worker is in HIR form and unknown to the dynamic class loading mechanism, it is compiled after creation. This work included making the optimising compiler re-entrant.
4. **removal of original loop body** - the head and exit blocks have their instructions removed and the head branches to

IEEE COMPUTER SOCIETY

the exit (assuming they're not the same block). As with loop duplication original variables within the loop are retained so that dependent instructions don't need altering.

5. **add code to copy invariants to a loop worker instance** - the invariant fields of the loop body need copying to a new instance of the loop worker.
6. **add call to spawn loop worker threads** - the final part of the new parallel loop body is to call a method responsible for forking and joining loop workers. The arguments to the thread spawning method are the loop worker, the initial loop iterator value and the terminal loop iterator value.

Figures 6 illustrates the operation of the new parallel loop body.

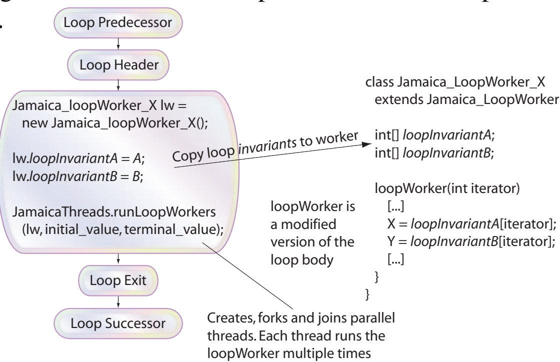

Figure 6. Creating the loop worker, assigning invariant values and calling thread distribution mechanism

4. Performance Analysis

We have benchmarked the potential optimisation using the Spec JVM benchmark suite [14]. To test the parallelisation optimisation we used a dual CPU Intel Pentium 4 computer running Linux 2.6.8, and with the Jikes RVM configured to use 2 underlying processors (VM_Processors/pthreads). The parallelisation optimisation parallelises the Jikes RVM, the class libraries and the benchmarks. By monitoring where loops were parallelised we determined that 250 were parallelised within the class library and the Jikes RVM upon creating its boot image, and 1 loop was parallelised while benchmarking _201_compress. We ran the benchmark in three configurations to determine our results:

1. standard Jikes RVM configuration,
2. Jikes RVM, parallelising loops and with spawning threads but not executing loop bodies in parallel,
3. Jikes RVM parallelising loops, spawning threads and executing loop bodies in parallel.

We present our results in Table 2. The speed-up is a measure of how much performance is gained running the DOALL loops in parallel on two processors. The speed-up is determined by comparing configurations 2 and 3. The overhead of thread creation is the time difference between running configuration 2 and configuration 1. The results for configuration 1 are presented as the normal benchmark time. Finally we show how many parallel loop bodies are executed during the running of each benchmark.

Overall an average speedup of 1.9% is shown by executing loop bodies in parallel. Figure 7 shows the performance normalised against the existing performance without any threading optimisation or thread creation overhead.The parallelization

Benchmark	Parallelisation speed-up[1]	Overhead	Normal benchmark execution time	Executed parallel loop bodies
_201_compress	1.6%	10.5s	7.1s	3500
_202_jess	0.7%	3.8s	3.5s	1500
_205_raytrace	3.7%	3.0s	4.3s	2400
_209_db	0.9%	8.4s	11.5s	4500
_213_javac	2.1%	0.2s	5.1s	1200
_222_mpegaudio	2.7%	11.9s	7.2s	12500
_227_mtrt	2.6%	1.7s	5.0s	1200
_228_jack	1.1%	1.3s	5.6s	1800

Table 2. Performance and overhead of parallelisation on the Spec JVM benchmarks

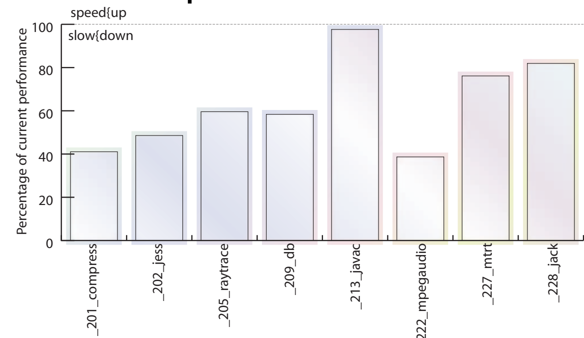

Figure 7. Parallel result normalised against current Jikes RVM performance

overheads slow the benchmarks down considerably, with the new performance being in the worst case 39% of the existing performance (or a 2.48 times slow-down). The overhead correlates to the number of parallel loop bodies executed and therefore the number of threads created. To avoid such large overheads it is possible to use a lightweight threading mechanism, and this is a subject of our on going compiler and architectural research. Similar research is being carried out for the open research compiler, where thread spawning costs are reported as being as low as 5 cycles [13].

To ensure that loops were indeed being parallelised and that in an idealised situation we could achieve performance speedups proportional to the number of processors, we implemented a small kernel benchmark shown in Figure 8.

On this test loop a speed-up of 45.2ms was caused by enabling the parallel threads. The total execution time of the main thread was 102.1ms. Ignoring threading costs, this is only a 79% speed-up on a dual processor machine. We currently believe that for the fine grain threads the thread scheduling mechanism of the Jikes RVM accounts for the performance loss.

We would aim to increase the number of parallel loop bodies executed and to improve the size of the loop bodies executed in parallel. A problem with the loop body size is partly caused by the pre-parallelisation optimisation phase presented in Section 2. It is only successful in removing null and bound checks from nodes in the annotated LST where the check is related to the loop iterator, thereby creating an unparallelisable loop with guard checks present inside outer loops. The Jikes RVM already eliminates many null and bounds check operations by propagating non-nullness and size information about arrays created within a method [5]. The potential of this is expanded greatly

[1] This is the performance speed-up excluding overheads introduced by creating threads and performing the optimisation.

IEEE COMPUTER SOCIETY

```
int size = 3000;
double[] matrix1 = new double[size];
double[] matrix2 = new double[size];
double[] result = new double[size];
for (int i = 1; i <= 500; i ++) {
    for (int p = 0; p < size; p ++) {
        matrix1[p] = p * p / i;
        matrix2[p] = p * (p + 1) / i;
        result[p] = (i * p + 1) / i;
    }
}
```

Figure 8. Simple loop to parallelise

by method inlining. In the situations where the array size and non-nullness isn't known our optimization exposes parallelism, but this could be done better by duplicating entire loops rather than working recursively.

5. Discussion

This paper has demonstrated that runtime parallelisation can achieve a speedup but only by negating the cost of the analysis and thread creation costs, of which the latter is by far the greater. However, does runtime parallelisation, shown here in a JVM, make sense? The following are the reasons that make runtime parallelisation currently desirable:

1. **simplicity in implementation** - having optimisations performed safely means programmers can concentrate on writing correct code rather than high-performance code.
2. **standard parallel back-end** - higher-level languages, such as those concentrating on Mathematics, are able to make greater assumptions as to how data is being used. These compilers can generate platform independent Java code, and by making these loops amenable to parallelisation optimisations within the JVM they can achieve parallel performance.

The downside is the cost in the compiler of performing this optimisation, but it is expected work on adaptive compilation will make this cost small.

A further advantage of the approach is that it should be adaptable to speculative threading. A speculative thread is one whose effects on memory can be thrown away (*squashed*). This is achieved by hardware buffering or by a modification to the cache. A speculative adaptation to our approach is to allow loops with break-out paths to be parallelised assuming the break-out path is never executed. If the break-out path is executed then the thread, and threads sequentially later than this one, can have their effects on memory squashed.

Such optimisations appear desirable but does this mean that dynamic compilers should have all the optimisations available in a static parallelising supercompiler? It seems that the intermediate form of Java (or other binary codes as considered in our work [10]) impose a great number of restrictions on what optimisations can be safely performed, and hence require expensive compile time analysis and run-time checks. It therefore seems likely that, for example, complex math codes should be written in higher-level languages with supercompiler restructuring optimisations, and the platform independent virtual machine layer only perform simple parallelisation optimisations where it has specific knowledge of the underlying hardware.

6. Summary and Conclusions

We have presented a runtime compiler, DOALL, thread-oriented, parallelisation optimisation and Java null and bound check optimisation to facilitate this. The optimisation achieves a speed-up over serially executed loop bodies on a dual processor computer of 1.9% on the SpecJVM benchmark suite, and on a small kernel benchmark a speed-up of 79%. The cost of threading on current computer architectures and in the standard JVM threading mechanism is higher than is suitable for this optimisation. We consider this work to be a first step toward the likely future dynamic compilers for future parallel computer architectures.

References

[1] The Jamaica project. http://www.cs.manchester.ac.uk/apt/projects/jamaica, May 2005.

[2] A. J. Bik, M. Girkar, P. M. Grey, and X. Tian. Efficient exploitation of parallelism on Pentium III and Pentium 4 processor-based systems. *Intel Technology Journal*, February 2001.

[3] A. J. C. Bik, J. E. Villacis, and D. B. Gannon. javar: A prototype Java restructuring compiler. *Concurrency: Practice and Experience*, 9(11):1181–1191, 1997.

[4] O. A. R. Board. OpenMP: Specifications. http://www.openmp.org/specs/, 2003.

[5] R. Bodik, R. Gupta, and V. Sarkar. ABCD: eliminating array bounds checks on demand. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 321–333, 2000.

[6] R. Choy and A. Edelman. Parallel MATLAB: doing it right. *Proceedings of the IEEE*, 93(2):331–341, February 2005.

[7] High Performance Fortran Forum. High Performance Fortran language specification, version 1.0. *Scientific Programming*, 2(1–2):1–170, June 1993.

[8] IBM. Jikes™ Research Virtual Machine (RVM). http://jikesrvm.sourceforge.net/, 2005.

[9] K. Kennedy and J. R. Allen. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[10] R. Matley. Native code execution within a JVM. Master's thesis, The University of Manchester, September 2004.

[11] S. S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[12] D. Naishlos. Autovectorization in GCC. In *GCC & GNU Toolchain Developers' Summit*, pages 105–117. IBM Research Lab in Haifa, June 2004.

[13] C. Quinones, C. Madriles, J. Sanchez, P. Marcuello, A. Gonzalez, and D. Tullsen. Mitosis compiler: An infrastructure for speculative threading based on pre-computation slices. In *SIGPLAN Conference on Programming Language Design and Implementation*, 2005.

[14] SPEC JVM98 benchmarks. http://www.spec.org/osg/jvm98/, 1998.

[15] M. J. Wolfe. The definition of dependence distance. *ACM Transactions on Programming Languages and Systems*, 16(4):1114–1116, July 1994.

[16] M. J. Wolfe. *High performance compilers for parallel computing*. Addison-Wesley, Redwood City, CA, 1996.

[17] J. Zhao, I. Rogers, and C. Kirkham. A system for runtime loop optimisation in the Jikes RVM. In *Postgraduate Research Conference in Electronics, Photonics, Communications and Networks, and Computing Science*, Lancaster, UK, March 2005.

0-7695-2405-2/05 $20.00 © 2005 IEEE

IEEE COMPUTER SOCIETY